Matrix-Chain Multiplication

Given: "chain" of matrices $(A_1, A_2, \ldots A_n)$, with A_i having dimension $(p_{i-1} \times p_i)$.

Goal: compute product $A_1 \cdot A_2 \cdots A_n$ as quickly as possible

Multiplication of $(p \times q)$ and $(q \times r)$ matrices takes pqr steps

Hence, time to multiply two matrices **depends on dimensions!**

Example:: $n = 4$. Possible orders:

$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Suppose A_1 is 10×100, A_2 is 100×5, A_3 is 5×50, and A_4 is 50×10

Order 2:

$$100 \cdot 5 \cdot 50 + 100 \cdot 50 \cdot 10 + 10 \cdot 100 \cdot 10 = 85,000$$

Order 5:

$$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 10 = 12,500$$

But: the number of possible orders is exponential!

We want to find **Dynamic programming** approach to **optimally** solve this problem

The four basic steps when designing DP algorithm:

1. **Characterize structure** of optimal solution
2. Recursively **define value** of an optimal solution
3. **Compute value** of optimal solution in bottom-up fashion
4. **Construct optimal solution** from computed information

1. Characterizing structure

Let $A_{i,j} = A_i \cdots A_j$ for $i \leq j$.

If $i < j$, then any solution of $A_{i,j}$ must split product at some k, $i \leq k < j$, i.e., compute $A_{i,k}$, $A_{k+1,j}$, and then $A_{i,k} \cdot A_{k+1,j}$.

Hence, for some k, cost is

- cost of computing $A_{i,k}$ plus
- cost of computing $A_{k+1,j}$ plus
- cost of multiplying $A_{i,k}$ and $A_{k+1,j}$.

Optimal (sub)structure:

- Suppose that optimal parenthesization of $A_{i,j}$ splits between A_k and A_{k+1}.
- Then, parenthesizations of $A_{i,k}$ and $A_{k+1,j}$ must be optimal, too (otherwise, enhance overall solution — subproblems are independent!).
- **Construct optimal solution:**
  1. split into subproblems (using optimal split!),
  2. parenthesize them optimally,
  3. combine optimal subproblem solutions.

2. Recursively def. value of opt. solution

Let $m[i,j]$ denote **minimum number of scalar multiplications** needed to compute $A_{i,j} = A_i \cdot A_{i+1} \cdots A_j$ (full problem: $m[1,n]$).

Recursive definition of $m[i,j]$:

- if $i = j$, then

  $$m[i,j] = m[i,i] = 0$$

  ($A_{i,i} = A_i$, no mult. needed).

- if $i < j$, assume optimal split at k, $i \leq k < j$. $A_{i,k}$ is $p_{i-1} \times p_k$ and $A_{k+1,j}$ is $p_k \times p_j$, hence

  $$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j.$$

- We do not know optimal value of k, hence

  $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] & \text{if } i < j \\ +p_{i-1} \cdot p_k \cdot p_j\} & \end{cases}$$

We also keep track of optimal splits:

$$s[i,j] = k \iff m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

3. Computing optimal cost

Want to compute $m[1,n]$, minimum cost for multiplying $A_1 \cdot A_2 \cdots A_n$.

Recursively, according to equation on last slide, would take $\Omega(2^n)$ (subproblems are computed over and over again).

However, if we **compute in bottom-up fashion**, we can reduce running time to poly(n).

Equation shows that $m[i,j]$ depends only on **smaller subproblems:** for $k = 1, \ldots, j-1$,

- $A_{i,k}$ is product of $k - i + 1 < j - i + 1$ matrices,
- $A_{k+1,j}$ is product of $j - k < j - i + 1$ matrices.

Algorithm should fill table m using increasing lengths of chains.

The Algorithm

```
n ← length[p] − 1
for i ← 1 to n do
    m[i, i] ← 0
end for
for ℓ ← 2 to n do
    for i ← 1 to n − ℓ + 1 do
        j ← i + ℓ − 1
        m[i, j] ← ∞
        for k ← i to j − 1 do
            q ← m[i, k] + m[k + 1, j] + p_{i−1} · p_k · p_j
            if q < m[i, j] then
                m[i, j] ← q
                s[i, j] ← k
            end if
        end for
    end for
end for
```

Example

A_1 (30 × 35), A_2 (35 × 15), A_3 (15 × 5), A_4 (5 × 10), A_5 (10 × 20), A_6 (20 × 25)

Recall: multiplying A $(p \times q)$ and B $(q \times r)$ takes $p \cdot q \cdot r$ scalar multiplications.

j \ i	1	2	3	4	5	6
6						0
5					0	
4				0		
3			0			
2		0				
1	0					

Example

A_1 (30×35), A_2 (35×15), A_3 (15×5), A_4 (5×10), A_5 (10×20), A_6 (20×25)

Recall: multiplying A $(p \times q)$ and B $(q \times r)$ takes $p \cdot q \cdot r$ scalar multiplications.

j \ i	1	2	3	4	5	6
6	15,125	10,500	5,375	3,500	5,000	0
5	11,875	7,125	2,500	1,000	0	
4	9,375	4,375	750	0		
3	7,875	2,625	0			
2	15,750	0				
1	0					

4. Constructing optimal solution

Simple with array $s[i, j]$, gives us optimal split points.

Complexity

We have three nested loops:

1. ℓ, length, $O(n)$ iterations
2. i, start, $O(n)$ iterations
3. k, split point, $O(n)$ iterations

Body of loops: constant complexity.

Total complexity: $O(n^3)$

All-pairs-shortest-paths

- Directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, $|V| = n$
- Weight of path $p = (v_1, v_2, \ldots, v_k)$ is $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$
- Assume G contains no negative-weight cycles
- **Goal:** create $n \times n$ matrix of shortest path distances $\delta(u, v)$, $u, v \in V$
- **1st idea:** use single-source-shortest-path alg (i.e., Bellman-Ford); but it's too slow, $O(n^4)$ on dense graph

Adjacency-matrix representation of graph:

- $n \times n$ adjacency matrix $W = (w_{ij})$ of edge weights
- assume

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

In the following, we only want to compute lengths of shortest paths, not construct the paths.

Dynamic programming approach, four steps:

1. Structure of a shortest path: Subpaths of shortest paths are shortest paths.

Lemma. Let $p = (v_1, v_2, \ldots, v_k)$ be a shortest path from v_1 to v_k, let $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ for $1 \le i \le j \le k$ be subpath from v_i to v_j. Then, p_{ij} is shortest path from v_i to v_j.

Proof. Decompose p into

$$v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k.$$

Then, $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Assume there is cheaper p'_{ij} from v_i to v_j with $w(p'_{ij}) < w(p_{ij})$. Then

$$v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$$

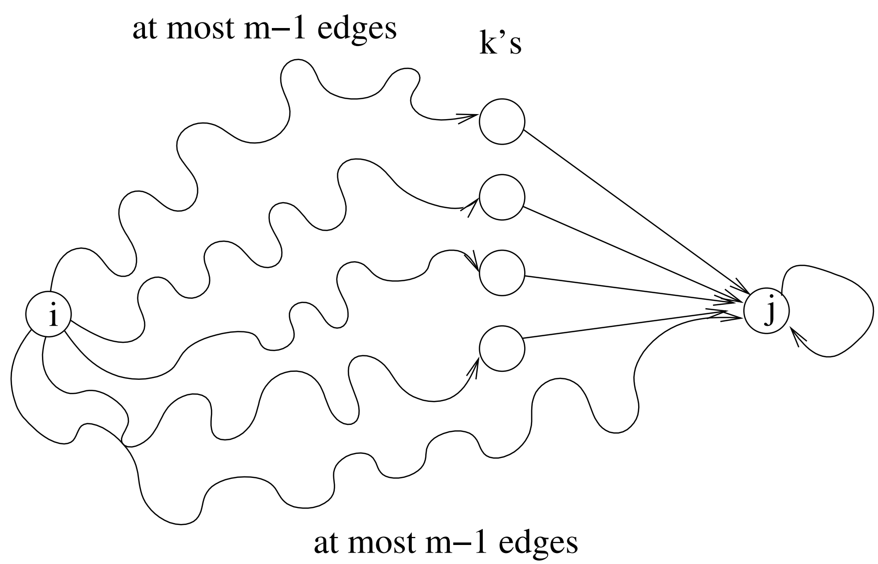
is path from v_1 to v_k whose weight $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, a contradiction.

2. Recursive solution and 3. Compute opt. value (bottom-up)

Let $d_{ij}^{(m)}$ = weight of shortest path from i to j that uses at most m edges.

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

$$d_{ij}^{(m)} = \min_k \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$



We're looking for $\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots$

Alg. is straightforward, running time is $O(n^4)$ ($n-1$ passes, each computing n^2 d's in $\Theta(n)$ time)

Unfortunately, no better than before...

Approach is similar to **matrix multiplication:**

$C = A \cdot B$, $n \times n$ matrices, $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$, $O(n^3)$ operations

Replacing "$+$" with "min" and "$\cdot$" with "$+$" gives

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\},$$

very similar to

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + w_{kj}\}$$

Hence $D^{(m)} = D^{(m-1)}$ "$\times$" W.

Floyd-Warshall algorithm

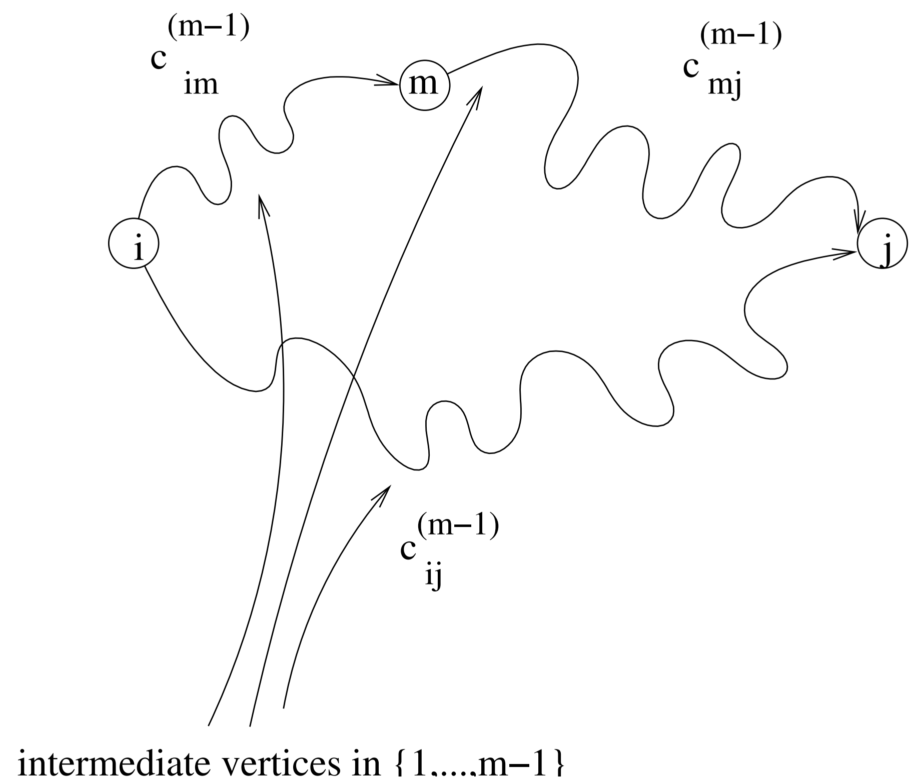
Also DP, but faster (factor $\log n$)

Define $c_{ij}^{(m)}$ = weight of a shortest path from i to j with **intermediate vertices** in $\{1, 2, \ldots, m\}$.

Then $\delta(i,j) = c_{ij}^{(n)}$

Compute $c_{ij}^{(n)}$ in terms of smaller ones, $c_{ij}^{(<n)}$:

$$
\begin{aligned}
c_{ij}^{(0)} &= w_{ij} \\
c_{ij}^{(m)} &= \min\left(c_{ij}^{(m-1)},\ c_{im}^{(m-1)} + c_{mj}^{(m-1)}\right)
\end{aligned}
$$



Difference from previous algorithm: needn't check *all* possible intermediate vertices. Shortest path simply either includes m or doesn't.

Pseudocode:

for $m \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      if $c_{ij} > c_{im} + c_{mj}$ **then**
        $c_{ij} \leftarrow c_{im} + c_{mj}$
      end if
    end for
  end for
end for

Superscripts dropped, start loop with $c_{ij} = c_{ij}^{(m-1)}$, end with $c_{ij} = c_{ij}^{(m)}$

Time: $\Theta(n^3)$, simple code

Best algorithm to date is $O(V^2 \log V + VE)$

Note: for dense graphs ($|E| \approx |V|^2$) can get APSP (with Floyd-Warshall) for same cost as getting SSSP (with Bellman-Ford)! ($\Theta(VE) = \Theta(n^3)$)